NPS-CS-10-009

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Putting Order Into the Cloud:
Object-oriented UML-based Rule Enforcement
for Document and Application Organization

by

D. Drusinsky, J.B. Michael, T.W. Otani and M. Shing

September 2010

Approved for public release; distribution is unlimited

Prepared for: Office of the DoD Chief Information Officer
1851 S. Bell St., Suite 600
Arlington, VA 22202

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

This report was prepared for and funded by the Office of the DoD CIO.

This report was prepared by:

_____

Doron Drusinsky
Associate Professor of Computer Science
Naval Postgraduate School

Reviewed by:

_____

Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____

Karl A. van Bibber
Vice President and Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
20 Sep 2010	Technical Report	Jan 18 – Sep 30, 2010

4. TITLE AND SUBTITLE
Putting Order Into the Cloud: Object-oriented UML-based Rule Enforcement for Document and Application Organization

5a. CONTRACT NUMBER
DWAM00390

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Doron Drusinsky, James Bret Michael
Thomas W. Otani, Man-Tak Shing

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Naval Postgraduate School
1411 Cunningham Road,
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-10-009

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of DoD Chief Information Officer
1851 S. Bell Street
Suite 600
Arlington, VA 22202

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT
Cloud computing describes a new distributed computing paradigm for IT data and services that involves over-the-Internet provision of dynamically scalable and often virtualized resources. While cost reduction and flexibility in storage, services, and maintenance are important considerations when deciding on whether or how to migrate data and applications to the cloud, large organizations like the Department of Defense need to consider the organization and structure of data on the cloud and the operations on such data in order to reap the full benefit of cloud computing. This report describes how object-oriented design using the UML, in addition to providing source control tools tailored for use in the cloud, can provide effective management of contents in the cloud, or what we term cloud control.

15. SUBJECT TERMS
Cloud computing, object and content management, document sharing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Doron Drusinsky
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclass-ified	Unclass-ified	Unclass-ified	UL	34	831-656-2168

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cloud computing describes a new distributed computing paradigm for IT data and services that involves over-the-Internet provision of dynamically scalable and often virtualized resources. While cost reduction and flexibility in storage, services, and maintenance are important considerations when deciding on whether or how to migrate data and applications to the cloud, large organizations like the Department of Defense need to consider the organization and structure of data on the cloud and the operations on such data in order to reap the full benefit of cloud computing. This report describes how object-oriented design using the UML, in addition to providing source control tools tailored for use in the cloud, can provide effective management of contents in the cloud, or what we term cloud control.

THIS PAGE INTENTIONALLY LEFT BLANK

1. Introduction

Most personal and business documents are organized in an ad hoc manner. Consider for example, a person who organizes his or her personal bank statements in folders, with each folder containing sub-folders labeled by year, each of which contains two subfolders named *checking* and *savings*, with twelve checking and twelve savings bank statement PDF files in each, respectively. In addition, he or she may have a separate folder named *taxes*, which also has subfolders labeled by year, each subfolder containing an electronic version of the corresponding annual tax return as well as scanned copies of receipts pertaining to allowed deductions such as mortgage interest payments and property tax statements. Clearly, in the event of an audit, this person will need to recreate the ad hoc relationships between those respective folders and files. For instance, the person being audited may be requested to supply all bank statements and receipts associated with the last three annual tax returns. Given the ad hoc organization of the documents the person must rely on his or her memory and manual navigation when traversing such relationship associations. Such an approach does not scale up for large organizations such as the Department of Defense (DoD) with approximately 3.5 million users who need the ability to share documents located in the cloud.

Contemporary object-oriented (OO) modeling uses the standard Unified Modeling Language (UML) to model classical OO principles of encapsulation, inheritance, and polymorphism as well as entity relationships (http://en.wikipedia.org/wiki/Unified_Modeling_Language). The major benefits of OO modeling, design and subsequent implementation are reusability, reliability, robustness, extensibility, and maintainability. After almost a quarter of a century of worldwide experience, the success of the OO and UML-based software engineering approach is widely accepted.

In the practice of modern software engineering, data is represented as objects and the entities responsible for performing operations on those objects are methods. OO encapsulation means that operations can only operate on designated properties of designated types of objects. The principle of encapsulation, now a key part of the prevailing OO software engineering approach, was a significant departure from the state of the art in the 1950's to the 1980's, when programming languages such as C allowed operations (C functions) to access data structures with few constraints other than the programmers common sense.

In this paper, we use the term *document objects* to refer to data objects which are documents. In the document object world - in which most lay computer users operate, applications are the entities that perform operations on document objects. Most contemporary platforms allow applications almost unlimited access to document objects. For example, it is not uncommon to hear an end user complain that an Excel document was corrupted after he or she edited it with Wordpad or Notepad and inadvertently changed an XML tag. Similarly, an end user can inadvertently change the total dollar amount field of a cost proposal instead of the labor field of the same proposal.

This report presents a technique for the specification and enforcement of relationships between document objects as well as relationships between document objects and the different applications using software engineering based Object-oriented (OO) modeling and enforcement.

The transition from a desktop-centric work environment to a cloud-centric environment provides an opportunity to rethink the way document objects and applications are organized, in addition to the way they are stored and delivered.

Source control is an additional popular and relevant software engineering technology that has been in use for more than two decades. In this paper we propose using a similar approach, called *cloud control* (CC), to control objects and applications on the cloud. In this report, we explain how CC can be used as an integral part of an enforcement scheme for the recommended document and application organization approach.

2. Object-Oriented Document and Application Organization

2.1 A Financial Documents Example

Consider the financial document organization example depicted in Figure 1. It consists of three *abstract* classes, *Account*, *PersonalAcct* and *BusinessAcct*. Being abstract means, in our context, that no document object exists for these classes. For example, from the document point of view, there is no explicit file or folder object representation for a bank account on the file system; information about bank accounts (e.g., name and SSN that are defined in the *Account* and *PersonalAcct* classes and inherited into the *PMonthlyStmt* and *PTaxReturn* classes) are captured in the files corresponding to the monthly statements and tax returns instead. There are two kinds of accounts, personal accounts and business accounts. A business account is related to one to five personal accounts of the business directors, and a personal account can be associated with any number of business accounts. A personal account contains two different types of documents (*PMonthlyStmt* and *PTaxReturn*) while a business account contains three different types of documents (*BMonthlyStmt*, *BTaxReturn*, and *ExpenseReport*). An expense report must be associated with a monthly statement for a business account and each monthly statement may be associated with a tax return document for either personal or business accounts. An expense report consists of a plurality of expense elements, where an element is a part of a document object. Like an abstract class, no document object exists for an *element* class; an element object contains data that are elements of some other objects. A tax return document may be associated with the tax return document of the previous year. In addition to these relationships, classes contain properties and methods, such as *PMonthlyStmt* class containing a *getExpenses*() method.

The UML diagram of Figure 1 is but a visual depiction of a schema; its XML schema counterpart is listed in Appendix A. Instances of the document and application objects that satisfy a schema are called *aggregate documents*. As with programming

counterparts, the cloud can contain a large plurality of aggregate documents that satisfy a given schema. Clearly, this empowers organizations to formally specify document ordering schemas rather than relying on ad hoc end-user organization instances, or relying on the end user to comply with and manually implement some written specification of a mandated document organization schema.

To this end, documents must contain clearly marked property and association segments. Contemporary document objects often contain such information: they are often Extensible Markup Language (XML) documents whose elements are effectively properties, and they often contain links to other documents, for example using a Uniform Resource Identifier (URI) or Uniform Resource Locator (URL) format. Associations can be implemented as embedded URIs or as external information stored by the CC discussed in the Section 5. Absent in contemporary document objects however is a specification of the applications that are permitted to operate on those object types and the specific properties that can be read from and written to.

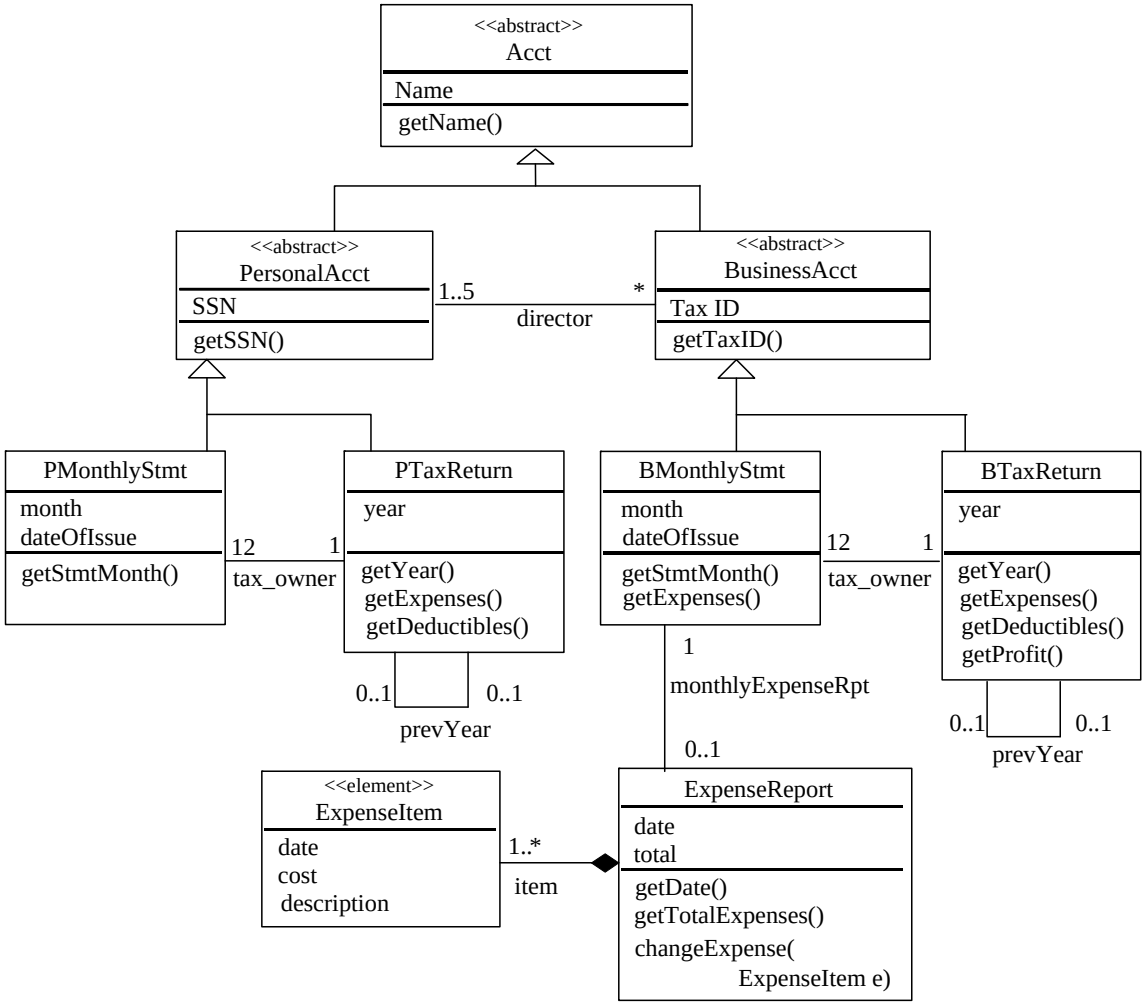


Figure 1. Class diagram schema for banking

2.2 Adaptation of Classical OO Principles to Document Objects and Applications

The three defining properties of classical OO programming (OOP) are inheritance, encapsulation, and polymorphism. In this section we examine their adaptation to the world of document objects and applications.

In OOP, *inheritance* is a way to form new classes (instances of which are called objects) using classes that have already been defined. Inheritance provides for reusing existing code with little or no modification. In our context, document objects may enjoy the inheritance principle in a similar manner. Consider a cloud repository with two types of documents, *checking document* (*CheckingDoc*) and *savings document (SavingsDoc)*. They are both special types of a *banking document (BankingDoc)*, as depicted in the class diagram of Figure 2. The *BankingDoc* parent has properties for the business name, date of creation, and account status. Given that most present day documents are stored in XML format, one can expect to locate those properties in the corresponding files; for instance, a *BankingDoc* has an XML element or elements that pertain to the abovementioned properties. Hence, a *SavingsDoc* virtually contains a name, date of creation, and status although these properties are not present in the *SavingsDoc* file; rather, their existence is deduced using the inheritance relationship. Note that we might expect to find a mismatch between the namespaces used by documents and those used by the schema, this issue should be resolved by the discovery step discussed below.

In the most general case, all child documents (i.e., checking and savings type documents) will therefore virtually contain property fields exposed by a parent (UML exposure is using a classifier such as *protected* or *public*) although they do not exist verbatim in the respective files. In such a situation, when applying a *changeStatus* application to a checking document, it changes the *status* property of its *parent* banking document. A *private* qualification in the parent is also useful in our OO cloud interpretation, meaning that no application that is applied to a child document can access the respective parent property field.

Child documents can obviously have their own fields and corresponding applications. For example, a savings document in Figure 2 is specified to allow a *readAsSavings* application, one that we envision highlights all information that pertains to the *interestRate* property.

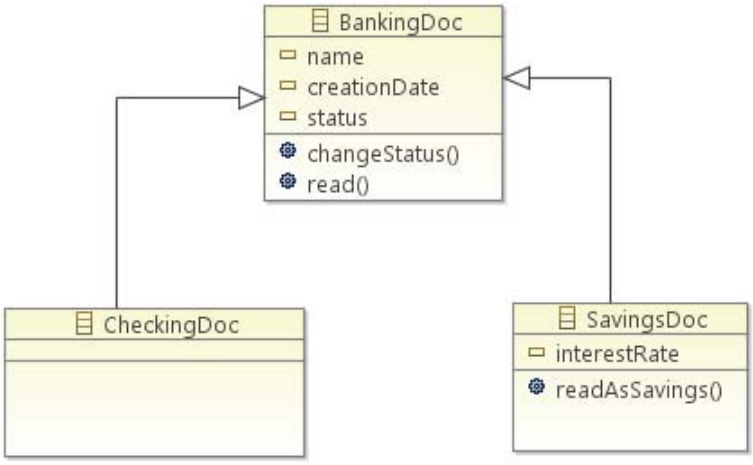


Figure 2. Inheritance in a business banking document repository

In OO programming, *encapsulation* refers to the encapsulation of properties and operations so that their interactions become clearly identified. For example, the Java class of Listing 1 contains two properties and an operation that operates on those properties only.

```
Class Foo {
  int x,y;

  …
  int qsum() {
    return x*x+y*y ;
  }
}
```

Listing 1. Encapsulation example in Java.

To this end, the financial documents example of Figure 2 manifests encapsulation, as in the following examples:

- A *readAsSavings* application can only be applied to a savings document.
- The *BankingDoc* parent also permits *a changeStatus* application access, namely, an application that changes the status property is allowed access to a banking document, that is, to any *CheckingDoc* and *SavingsDoc* type document; no other application access is specified as allowed.

Clearly one needs to define a convention for identifying the properties a given application is allowed to operate on. For example, one possible convention is to use the application name *changeStatus* to imply that that application can only affect the *status*

field/property. A different approach is to use the formal arguments in the UML diagram, such as *changeExpense(ExpenseItem e)* of Figure 1 meaning that the *changeExpense* application can only modify the *ExpenseItem* field of the ExpenseReport document; we can further restrict the *changeExpense* application's access to the *ExpenseItem* field to read-only with the formal argument *changeExpense(final ExpenseItem e).*

Note that encapsulation in OO programming is a human modeling and design concept, not necessarily an implementation feature. An OO program, such as the Java class in Listing 1 realizes the object as a data structure (the x,y values) on the heap, but the *qsum* code actually executes elsewhere—in a thread. Likewise, a cloud user can benefit from encapsulation as an organizational and control feature, yet implementation can still keep data objects and applications separate.

On OO programming, *subtype polymorphism*, also universally called just polymorphism, is the ability of one type, *A*, to appear as and be used like another type, *B*. The same principle applies well to cloud documents. For example, consider a modification of the hierarchy of Figure 2 in which: (i) a banking document allows an *edit* application for the purpose of editing of a banking document's *name* field only, and (ii) a savings document also allows an *edit* operation for the purpose of modifying the interest rate in addition to the operations permitted on the parent. If the end user applies an *edit* application to a checking document it will (using inheritance) apply it using (i), whereas an *edit* application to a savings document will apply it using (ii).

3. Adaptation of Entity Relationships to Document Objects

Classical OO modeling (of inheritance, encapsulation, and polymorphism) was merged with entity-relationship (ER) diagramming to create the UML class diagram notation, such as in Figure 1.

Some examples of relationships that must exist in an aggregate document that conforms to the schema of Figure 1 are:

- An expense report document must contain at least one expense element.
- A tax return document must be associated with twelve monthly statement documents.
- A business monthly statement can contain an expense report, but no more than one.

The next section addresses the question of whether *unspecified* relationships are permitted to exist on the actual cloud.

4. Rule Enforcement

Enforcement is the automated activity of enforcing the document organization schema on instances of actual documents and applications. The enforcement engine must check *all* relationships and properties in the actual document set for conformance with the schema. For example, per the schema introduced in Figure 1, the enforcement engine will check that all relationships specified in Figure 1 are conformed to by the document

instances. Figure 1 specifies, for instance, that personal monthly bank statements (instances of *PMonthlyStmt* class, e.g., in PDF format) are associated with *PTaxReturn* object instances.

In addition, the enforcement engine must validate that instance documents on the cloud contain the properties specified by Figure 1. Moreover, the enforcement engine also restricts applications to operate only on certain data elements as specified by the schema; for example, *changeExpense* of the *ExpenseReport* class of Figure 1 is specified to operate only on instances of the *ExpenseItem* class (which is an *element* class, i.e., no verbatim ExpenseItem file needs to exist on the platform). According to the class diagram an *ExpenseItem* element must be a constituent element of an *ExpenseReport* document. Consequently *changeExpense* is prohibited from touching anything in any *ExpenseReport* file other than the *ExpenseItem* instances, in contrast with contemporary applications that are free to do whatever they elect to do on their target documents. With most cloud implementations being based on prevailing contemporary operating systems, this kind of enforcement is not readily available. One possible solution however is for an event-based enforcement engine, discussed below, to allow objects "checkout" only when the request is made by an application that satisfies the criteria set by the schema.

4.1 Schema Enforcement Scheduling

Two enforcement scheduling approaches come to mind: periodic and event-based. A periodic enforcement engine traverses the cloud much like a crawler traverses the Internet. When it discovers objects it validates their structure (i.e., are the objects' actual properties in compliance with the schema), and traverses associations to yet another object on the cloud, for which it repeats the abovementioned procedure. This process amounts to graph traversal with additional validation of object-structure in each node.

The drawbacks of a periodic enforcement scheduling approach is that it does not detect violations at the time an end user of the system takes actions that violate a schema. For example, per the schema of Figure 1, if an end user removes a *BMonthlyStmt* document then all links in associated *ExpenseReport* documents break, yet the discovery of this anomaly will probably be delayed.

An event-driven enforcement scheduling approach means that a schema is validated whenever a relevant event occurs, such as a user modifying, moving, copying, or renaming a document file. One possible event-driven enforcement approach makes good use of a CC system, as follows. Because the CC serves as a gateway to the cloud, no object or application really exists unless the CC says so. As with source control counterparts, every meaningful modification to the elements registered in the CC is first detected by the CC, allowing the enforcement engine to validate the integrity of the document set and its conformance to all schemas. If the validation fails, the action is disallowed; as with their source control counterparts, CC has the capability to rollback the state of the repository or even, in some implementations, block the violating action.

4.2 Multi-Schema Considerations

We envision environments in which a plurality of schemas are used and enforced. In such a context a single document type (e.g., class *BTaxReturn* of Figure 1) may be specified in more than one schema. The following are several interesting cases that could arise:

- *Multiple inheritance*. A document class *A* might be specified to extend class *B* in one schema and class *C* in another. There is no ill-defined relationship here; the specified inheritance relationships will be enforced and used when their corresponding schema is enforced.
- *Union of associations*. A document class *A* might have association *rel_1* with class *D* specified by one schema and association *rel_2* with class *E* specified by another schema. Even if the two relationships are different (e.g., $D \neq E$) then, again, the situation does not induce an ill-defined relationship; the specified associations will be enforced and used when their corresponding schema is enforced. However, a conflict might occur if $D = E$ and the respective quantification do not agree, such as one having *minOccurances* = 5 and the other having *maxOccurances* = 1. Clearly, the enforcement tool must prune out such contradictions when a schema is deployed.
- *Unspecified associations*. We suggest that if a document object has a relationship that does not conform to *any* schema then rule enforcement must identify this as an error.

We also envision a situation where hierarchical organizations will impose schemas in a hierarchical manner, much like state laws exist in addition to federal laws. To this end, an additional required feature is the ability of a schema to *disallow* certain associations. For example, an accounting firm might disallow direct associations between an expense report type document and a tax return document. Note that the XML approach of using *maxOccurences=0* constraint in an XML schema file is insufficient because it relates only to a specific association between two entities, not to all possible associations between that pair.

5. Cloud Control

The jury has been out for many years regarding the added value of source control to the software design and development process. The following aspects of source control lend themselves nicely to the cloud and CC environment.

- *Multiuser environments*. Software projects often have multiple developers. An object (e.g., a Java source code file) is often owned by a certain developer, but is often also developed by multiple simultaneous users.

We see a similar trend in the emerging cloud world; for example, multiple users owning and co-editing a shared Google Docs document for a Pot-Luck lunch organization.

- *Ownership*. More often than not, objects change ownership over time, as when the original developer leaves the company, so much so that ownership manifestation is often reduced to a list of *author* comments on the top of a source code file. In other words, a source-controlled document has a life of its own that persists beyond the lifespan of an individual employee or that of a certain machine. Source control is therefore the tool organizations use to locate objects. Document objects have a similar lifecycle, with CC providing the same lifecycle ownership support.
- *Storage*. The location of the source control repository can be independent of the location of an individual developer; in fact, source control systems cater well for a distributed development team work environment.

  Clearly, such flexibility is one of the most obvious benefits lay people see in the cloud approach.
- *Online vs. Offline usage*. Engineers tend to use source control in two modes: one is directly connected to the repository, and the other is offline, with frequent commits. Many source control tools support both modes of operation. The second approach is useful when working offline, and as a means for developing code and not committing it until tested or approved by a higher authority.

  Clearly, these use cases exist for many cloud users who are concerned about the availability of their data during network down times. One difference between source control and CC is the availability of applications, which is not supported by offline source control, yet we envision that offline cloud users will have access to some of their applications.
- *Visualization.* As illustrated in Figure 3, visualization provides clear interactive indication as to the status of every object, where a *marked* object (e.g., the *StatechartLicenseAdmin.java* file in Figure 3) means that its local version differs from the repository version. Similar visualization will also be beneficial to cloud users.

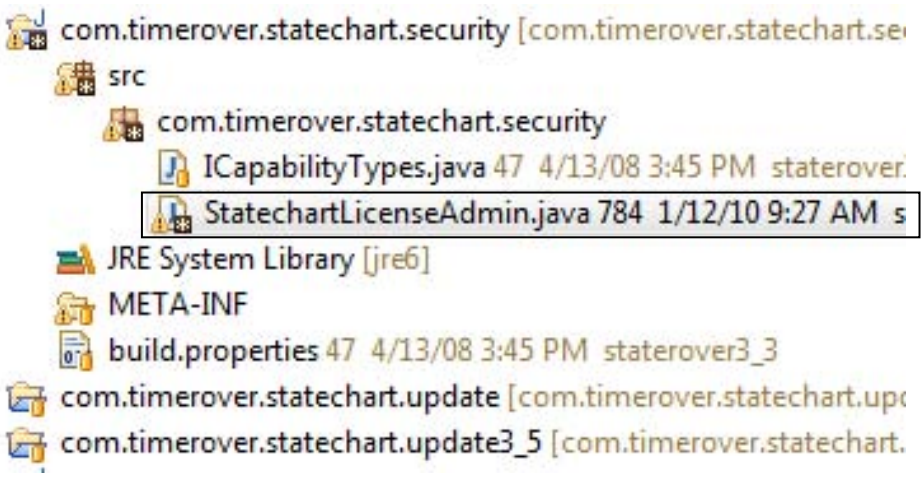


Figure 3. Subsclipse (Eclipse SVN plug-in) visual cue for a modified file

- *Version control*. Programmers need to be able to revert to old versions without being required to manually store and label them. They also need to be able to branch out multiple development branches (e.g., two teams working on two porting projects for a Windows application, one to Linux and the other to Mac). Similarly, cloud document authors often perform document branching.

5.1 Cloud Control versus Source Control

Source control, also known as version control or software configuration control, deals with the management of changes to documents, programs, and other information stored as computer files. Typical functions supported by a source control tool include:

- Management of the physical storage of files in the repository and in the working folders of individual developers.
- Management of the historical record of the changes made on the files over time via version numbers, labels and tags.
- Checkin/checkout mechanisms to control concurrent access to the files.
- Merge and branching mechanisms to allow developers to work on different parts of the software in parallel and merge their efforts later.

In addition to the aforementioned source control functions, any tool that supports cloud control should also provide functionalities to:

- Specify and enforce the schematic rules on the relationships between the different document objects.
- Specify and enforce the relationship between the document objects and the different applications for processing the objects.
- Create, modify, search, retrieve and destroy the document objects and relationships

6. Organization and Enforcement Benefits

Some benefits of the recommended OO organization and enforcement approach are the following.

6.1 Merging Documents

Consider an often-discussed use case for the cloud where a shared document *D* is used by a group on the cloud. An individual user then downloads *D*, and edits it on his or her desktop (a situation that might arise when connectivity is disrupted), thereby creating version *D′*. Later, this user wants to upload his or her version and merge it with the cloud version. This kind of use case is quite common in a software engineering setting in which users often work offline and then merge their code changes with the repository; contemporary source control tools provides ample tool support for such merge operations.

Merging documents that are serialized in binary or some other cryptic proprietary notation is clearly a nightmare. Fortunately, most contemporary applications use XML notation when serializing their documents. XML notation however, does not solve the problem because the merge operation is not well defined, as follows. First, typical XML tags in most office documents are not semantic; rather, they identify editing information such as lines, paragraphs, and fonts, information that cannot be used for semantic merging. Secondly, consider an example where both documents *D* and *D′* contain element *el*; should this case be pronounced as a conflict or should there be two *el* fields in the merged version? Clearly, if *el* is actually *date* then most readers would agree that that field should be unique, that is, the two instances of *el* should be merged into one. However, if *el* is actually *expense date* then *D* can contain a long list of expenses, i.e., both instances of *el* should be used in the merged document.

A schema such as in Figure 1 resolves these issues. If two *ExpenseReport* type documents are merged then the schema clearly specifies that an *ExpenseReport* can contain a plurality of *ExpenseItem* elements. Hence, the merge tool can simply create a union of all expenses in the merged *ExpenseReport.* We propose that the UML schema have a specification as to how to manage conflicts. For example, the schema also indicates that the *Date* field of an *ExpenseReport* is unique; hence if the dates in both merged documents do not agree, the merge tool will pick the most recent, or perhaps even the date of the merge, depending on the UML specification that pertains to a merge-conflict in that field.

6.2 Discovery

When a document is added to the cloud, with either enforcement scheduling approach discussed earlier, a necessary precursor to schema enforcement is discovery, in which we map an object to one or more schemas. The following mappings are suggested to discover and map an instance document or applications object to a schema:

1. The mapping of the document to schema *classes*; for example, per the schema of Figure 1, when a PDF file is added to the cloud we need to discover whether it is an instance of *PMonthlyStmt*, *PTaxReturn*, *BMonthlyStmt*, *ExpenseReport*, or *BTaxReturn.*
2. The mapping of the document to other documents according to the relationships described in the schema. For example, if the document discovered in step #1 is a *BMonthlyStmt* class of Figure 1 then we need to discover its associated *BTaxReturn* document, if it exists.
3. The mapping of the document's parts, components, or elements to the *properties* of the class discovered in step #1.
4. The mapping of applications to the *methods* specified for its class in the schema. For example, if the class discovered in step #1 is a *BTaxReturn* class document per the schema of Figure 1, then we need to discover the application that corresponds to *getExpenses* method of that class.

Note that this is a significant departure from the way operating systems typically view applications. While traditionally the domain and co-domain[1] of an application are mapped to one or more file objects, this mapping maps the domain and co-domain of the applications to particular components, parts, or elements of file objects as specified in mapping #3.

7. Workflow

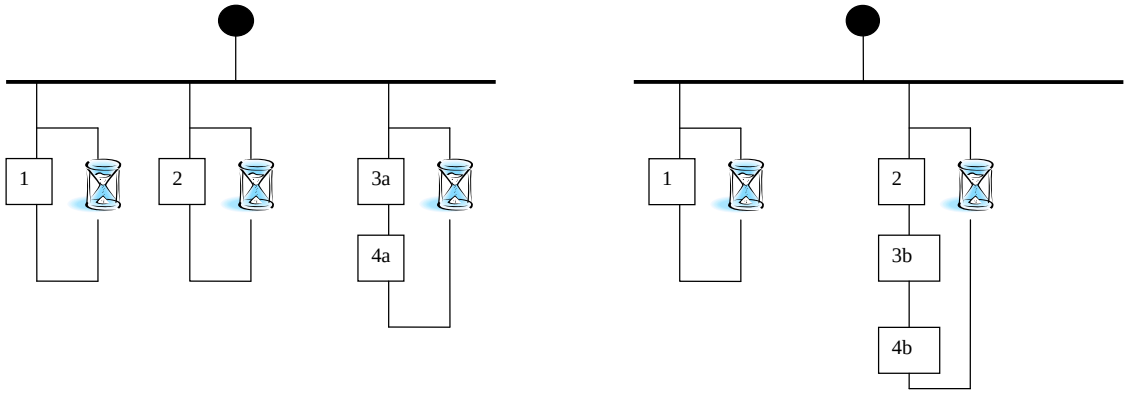
The workflow consists of four main tasks organized in two primary workflows, one for periodic scheduling and the other for event-driven scheduling. The two workflows are depicted in the UML Activity Diagrams (ADs) of Figure 4. The four tasks, denoted respectively as 1, 2, 3a, 4a in Figure 4(a) and 1, 2, 3b, 4b in Figure 4(b), are:

1. Perform OO design, typically as UML class diagrams, such as the one shown in Figure 1.
    a. Classic OO design: hierarchical design, property design, method specification (classical OO encapsulation), such as *PMonthlyStmt* being a subclass of *PersonalAcct* in Figure 1.
    b. Extended OO design: entity relationships in UML class diagrams, such as the relationship between *PMonthlyStmt* and *PTaxReturn* in Figure 1.
    c. Specify read/write permissions to operations, for example, *getExpenses*() of *BMonthlyStmt* object in Figure 1 being a read-only operation.
    d. Specify additional properties for merging.

[1] Let $f:A \rightarrow B$ be a function from A into B. The set A is called the domain of the function f, and B is called the co-domain of f.

The result of this step is a rule schema, with the prevailing representation being an XML Schema Definition (XSD).

2. Deploy objects and applications
    a. Deploy objects on platform. For example, an expense report spreadsheet file is deposited on the cloud or on operating system file system.
    b. Deploy applications on platform, for example, deploying an expense report browser on a cloud or operating system.
3. Discover objects and applications and map those artifacts to the rule schema
    a. Periodic discovery of objects and applications, as discussed in the Discovery section.
    b. Event-based discovery of objects and applications, as discussed in the Discovery section.
4. Schema enforcement
    a. Periodic enforcement, for example during "end of crawl session."
    b. Event-driven enforcement, for example, by enabling or disabling objects and applications to be checked in CC.



(a) Workflow when using periodic discovery and rule enforcement

(b) Workflow when using event based (using CC) discovery and rule enforcement

Figure 4. Activity diagram representation of workflow

The major difference between the two workflows is that there is a delay between task 2 and tasks 3 and 4 when periodic discovery and rule enforcement is used.

7. Prior State-of-Art

HyperText Markup Language (HTML) links provide a contemporary method for the specification of document-to-document and document-to-application links. While HTML links provide a reasonable technique for the implementation of links, one that can be adopted by a realization of the ideas presented in this report, this is but one aspect of the approach. In addition to the implementation of links our approach specifies and enforces the following:

- Whether a link must exist
- The plurality of links
- The induced association of a link to a document by virtue of inheritance (i.e., if a parent document has a protected or public association then a child document enjoys that link too).

In addition, HTML links do not cater for the OO principle of encapsulation.

The article: "Build an Object-oriented File System in PHP" (http://www.devx.com/webdev/Article/22240) describes a basic OO file system approach. It differs from this proposal in the following key elements:

- The article focuses on web sites, not on general file systems or the cloud.
- The article does not address the specification and enforcement of the object-relationship aspect of OO design and UML other than the inheritance relationship; for example it does not deal with entity relationship as between *BMonthlyStmt* and *BTaxReturn* in Figure 1.
- The article does not address the specification and enforcement of the OO encapsulation principle, that is, it does not address the manner in which applications are strictly associated with property elements of specific document types.

The patent "Object oriented file system in an object oriented operating system," United States Patent 5758153, also describes a basic OO file system approach. It differs from this proposal in the following key elements:

- The patent does not address the specification and enforcement of object relationships other than classical OO features of inheritance and encapsulation, namely it provides no method for the specification or general entity relationships (e.g., as expressed by a UML class diagram).
- The patent addresses the OO principle of encapsulation in a limited manner whereby the only supported operations are standard operations (such as create, open, close, and property accessors) and their variations, rather than any general-purpose application.

- In addition, the patent does not address the specification and enforcement of encapsulation, that is, the manner in which applications are restricted in their access to pre-specified elements of specific document types.
- The patent ignores the issue of enforcement of document object relationships and application to object relationships) because it does not cater for object-to-object relationships other than inheritance.

Moreover, OO operating systems, such as a Java-based OS, do not necessarily treat document objects and applications any differently than a C-based OS such as Unix. Specifically, the OO document object and application relationships specification and enforcement discussed in this paper are not addressed at all.

7. Conclusion

We have shown that using a universally understood formalism for specifying and enforcing extended OO relationships, both between documents and between applications and documents, is superior to the prevailing ad hoc method of achieving similar business rules. The current transition from a desktop- to a cloud-centric work environment provides an excellent opportunity to rethink not only how document objects and applications are stored and delivered, but also the way they are organized.

Some of the benefits of the proposed approach are:

- *Compositionality*. A complex aggregate document can be adequately described by a large collection of small documents organized in a well-defined manner.
- *Maintainability*. An aggregate document is easily updated by replacing small constituent document elements while keeping the induced aggregate document intact. Also, an application can be changed (e.g., from Microsoft PowerPoint to the OpenOffice version) with assurance that that the new version still operates on the intended fields of the intended object types.
- *Correctness and Accuracy*:
  - When updating an aggregate document (e.g., by updating or replacing constituent document objects), automatic rule enforcement maintains the correct relationships within the aggregate.
  - Applications are only permitted to operate on clearly specified elements of specified document types, thereby reducing the risk of inadvertent data corruption due to unintended consequences of execution of the application.
- *Traceability*. An integral part of the technology supporting OO rule-enforcement is the ability to traverse associations.
- *Sharing*:
  - An aggregate document schema can cross the boundaries of an individual user.

    - Document objects can be shared by more than one aggregate document (as long as the application specifications in all schemas are free of conflicts).

- *Portability.* When constituent objects are relocated, the enforcement rule automatically validates that the constraints of all aggregate document schemas is still maintained.

We showed that CC is a useful approach for enforcing document and document-application relationships on the cloud. In addition, CC provides a mechanism and gateway that can enforce security rules, and cater for versioning, traceability, and portability, in addition to ownership recording.

There are many related topics this report has not addressed, such as security. For example, when a document type specifies an application (e.g. *Foo.exe*) that can operate on a specific property, how does an application on the cloud authenticate itself as *Foo.exe*? This and other questions will need to be asked and answered to gain momentum for adoption of CC.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A. XML schema (XSD) for UML class diagram of Figure 1

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    xmlns:financialdocs="http://financialdocs/1.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" ecore:nsPrefix="financialdocs"
    ecore:package="financialdocs" targetNamespace="http://financialdocs/1.0">
<xsd:import namespace="http://www.eclipse.org/emf/2002/Ecore"
    schemaLocation="platform:/plugin/org.eclipse.emf.ecore/model/Ecore.xsd"/>
<xsd:element ecore:ignore="true" name="Acct" type="financialdocs:Acct"/>
<xsd:element ecore:ignore="true" name="PersonalAcct"
    type="financialdocs:PersonalAcct"/>
<xsd:element ecore:ignore="true" name="BusinessAcct"
    type="financialdocs:BusinessAcct"/>
<xsd:element ecore:ignore="true" name="PMonthlyStmt"
    type="financialdocs:PMonthlyStmt"/>
<xsd:element ecore:ignore="true" name="PTaxReturn"
    type="financialdocs:PTaxReturn"/>
<xsd:element ecore:ignore="true" name="BMonthlyStmt"
    type="financialdocs:BMonthlyStmt"/>
<xsd:element ecore:ignore="true" name="BTaxReturn"
    type="financialdocs:BTaxReturn"/>
<xsd:element ecore:ignore="true" name="ExpenseReport"
    type="financialdocs:ExpenseReport"/>
<xsd:element ecore:ignore="true" name="ExpenseItem"
    type="financialdocs:ExpenseItem"/>
<xsd:element ecore:ignore="true" name="AggregateDoc"
    type="financialdocs:AggregateDoc"/>
<xsd:complexType abstract="true" name="Acct">
    <xsd:attribute name="Name" type="ecore:EString"/>
    <xsd:annotation>
        <xsd:appinfo ecore:key="operations"
             source="http://www.eclipse.org/emf/2002/Ecore">
            <operation name="getName"/>
        </xsd:appinfo>
    </xsd:annotation>
</xsd:complexType>
<xsd:complexType abstract="true" name="PersonalAcct">
```

```
    <xsd:complexContent>
        <xsd:extension base="financialdocs: Acct">
            <xsd:sequence>
                <xsd:element ecore:resolveProxies="true" maxOccurs="5"
                    minOccurs="1" name="director" type="financialdocs:BusinessAcct"/>
            </xsd:sequence>
            <xsd:attribute ecore:name="SSN" name="SSN"
                type="ecore:EIntegerObject"/>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getSSN"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType abstract="true" name="BusinessAcct">
    <xsd:complexContent>
        <xsd:extension base="financialdocs: Acct">
            <xsd:attribute name="taxID" type="ecore:EIntegerObject"/>
            <xsd:attribute ecore:reference="financialdocs:PersonalAcct" name="director"
                use="required">
                <xsd:simpleType>
                    <xsd:restriction>
                        <xsd:simpleType>
                            <xsd:list itemType="xsd:anyURI"/>
                        </xsd:simpleType>
                        <xsd:maxLength value="5"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getTaxID"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="PMonthlyStmt">
    <xsd:complexContent>
        <xsd:extension base="financialdocs:PersonalAcct">
            <xsd:attribute name="month" type="ecore:EIntegerObject"/>
```

```
            <xsd:attribute name="dateOfIssue" type="ecore:EDate"/>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getStmtMonth"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="PTaxReturn">
    <xsd:complexContent>
        <xsd:extension base="financialdocs:PersonalAcct">
            <xsd:attribute ecore:name="tax_owner"
                ecore:reference="financialdocs:PMonthlyStmt" name="tax_owner">
                <xsd:simpleType>
                    <xsd:restriction>
                        <xsd:simpleType>
                            <xsd:list itemType="xsd:anyURI"/>
                        </xsd:simpleType>
                        <xsd:maxLength value="12"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="year" type="ecore:EIntegerObject"/>
            <xsd:attribute ecore:reference="financialdocs:PTaxReturn" name="prevYear"
                type="xsd:anyURI"/>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getYear"/>
                </xsd:appinfo>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getExpenses"/>
                </xsd:appinfo>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getDeductibles"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BMonthlyStmt">
```

```
    <xsd:complexContent>
        <xsd:extension base="financialdocs:BusinessAcct">
            <xsd:attribute ecore:reference="financialdocs:ExpenseReport"
                name="monthlyExpenseRpt" type="xsd:anyURI"/>
            <xsd:attribute name="dateOfIssue" type="ecore:EDate"/>
            <xsd:attribute name="month" type="ecore:EIntegerObject"/>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getStmtMonth"/>
                </xsd:appinfo>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getExpenses"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="BTaxReturn">
    <xsd:complexContent>
        <xsd:extension base="financialdocs:BusinessAcct">
            <xsd:attribute ecore:name="tax_owner"
                ecore:reference="financialdocs:BMonthlyStmt" name="tax_owner">
                <xsd:simpleType>
                    <xsd:restriction>
                        <xsd:simpleType>
                            <xsd:list itemType="xsd:anyURI"/>
                        </xsd:simpleType>
                        <xsd:maxLength value="12"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute ecore:reference="financialdocs:BTaxReturn" name="prevYear"
                type="xsd:anyURI"/>
            <xsd:attribute name="year" type="ecore:EJavaObject"/>
            <xsd:annotation>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getYear"/>
                </xsd:appinfo>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getExpenses"/>
                </xsd:appinfo>
```

```
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getDeductibles"/>
                </xsd:appinfo>
                <xsd:appinfo ecore:key="operations"
                    source="http://www.eclipse.org/emf/2002/Ecore">
                    <operation name="getProfit"/>
                </xsd:appinfo>
            </xsd:annotation>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ExpenseReport">
    <xsd:attribute name="date" type="ecore:EDate"/>
    <xsd:attribute name="total" type="ecore:EIntegerObject"/>
    <xsd:attribute ecore:reference="financialdocs:ExpenseItem" name="item">
        <xsd:simpleType>
            <xsd:list itemType="xsd:anyURI"/>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:annotation>
        <xsd:appinfo ecore:key="operations"
            source="http://www.eclipse.org/emf/2002/Ecore">
            <operation name="getDate"/>
        </xsd:appinfo>
        <xsd:appinfo ecore:key="operations"
            source="http://www.eclipse.org/emf/2002/Ecore">
            <operation name="getTotalExpenses"/>
        </xsd:appinfo>
        <xsd:appinfo ecore:key="operations"
            source="http://www.eclipse.org/emf/2002/Ecore">
            <operation name="changeExpense">
                <parameter name="e" type=ExpenseItem"/>
            </operation>
        </xsd:appinfo>
  </xsd:annotation>
</xsd:complexType>

<xsd:complexType name="ExpenseItem">
    <xsd:attribute name="date" type="ecore:EDate"/>
    <xsd:attribute name="cost" type="ecore:EFloatObject"/>
    <xsd:attribute name="description" type="ecore:EString"/>
</xsd:complexType>

<xsd:complexType name="AggregateDoc">
```

```
    <xsd:attribute ecore:name="BusinessStuff"
        ecore:reference="financialdocs:BusinessAcct" name="BusinessStuff">
        <xsd:simpleType>
            <xsd:list itemType="xsd:anyURI"/>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute ecore:name="PersonnalStuff"
        ecore:reference="financialdocs:PersonalAcct" name="PersonnalStuff">
        <xsd:simpleType>
            <xsd:list itemType="xsd:anyURI"/>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>

</xsd:schema>
```

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Peter Denning
   Naval Postgraduate School
   Monterey, California

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

6. Professor Bret Michael
   Naval Postgraduate School
   Monterey, California

7. Professor Thomas Otani
   Naval Postgraduate School
   Monterey, California

8. Professor Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

9. Mr. John Shea
   Office of the DoD Chief Information Officer
   Arlington, Virginia

10. COL Kevin Foster, USA
    Office of the DoD Chief Information Officer
    Arlington, Virginia

11. Professor George Dinolt
    Naval Postgraduate School
    Monterey, California

12. Professor Loren Peitso
Naval Postgraduate School
Monterey, California

13. Professor Scott Cote
Naval Postgraduate School
Monterey, California

14. Professor Albert Barreto
Naval Postgraduate School
Monterey, California

15. Mr. Alex Nelson
Naval Postgraduate School
Monterey, California

16. Mr. Scott J Dowell
Computer Science Corporation
San Diego, California

17. Mr. Michael Lee
Touchstone Consulting Group
Washington, D.C.

18. Ms. Karen Gordon
Institute for Defense Analyses
Alexandria, Virginia

19. Dr. Jeffrey Voas
National Institute of Standards and Technology
Gaithersburg, Maryland

20. Dr. Mark Lee Badger
National Institute of Standards and Technology
Gaithersburg, Maryland

21. Dr. Tim Grance
National Institute of Standards and Technology
Gaithersburg, Maryland